Evaluation des performances d'un noyau de simulation répartie

Philippe Ingels, Carlos Maziero

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1 9 9 2

25*ème anniversaire*

N° 1751

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

EVALUATION DES PERFORMANCES D'UN NOYAU DE SIMULATION REPARTIE

Philippe INGELS
Carlos MAZIERO

Septembre 1992

RR_1751

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tel. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Evaluation des Performances d'un Noyau de Simulation Répartie

Philippe INGELS, Carlos MAZIERO
IRISA – Campus de Beaulieu
35042 RENNES cedex – FRANCE
<*Name*>@irisa.fr

Programme 1, Projet ADP (Algorithmes Distribués et Protocoles)

Publication Interne n° 675 - Septembre 1992 - 36 pages

Résumé

L'utilisation du temps physique comme référence pour la synchronisation des processus d'une application répartie peut être étendue à une notion logique de temps : le temps virtuel. Dans un tel contexte, il existe une horloge virtuelle globale à l'ensemble des processus dont l'évolution est cadencée par des règles propres à l'application et non par un phénomène physique extérieur. Cette horloge permet d'organiser le calcul, de contrôler son l'évolution, de dater des événements, etc. Cet article présente l'implémentation et l'évaluation d'un noyau de système réparti appelé *Floria* et réalisé à l'IRISA; ce noyau permet d'utiliser la notion de temps virtuel dans une application. Dans un premier temps, le noyau et son interface sont présentés, et les principaux choix de mise en œuvre sont discutés. Ensuite, quelques exemples d'utilisation du noyau sont donnés. Finalement, les résultats des expériences effectuées pour évaluer les performances du prototype construit sont présentés et analysés.

Mots-clés: systèmes répartis, synchronisation, temps virtuel, simulation répartie, évaluation de performance.

Abstract
Performance Evaluation of a Kernel for Virtual Time Driven Applications

The use of physical time as a reference to synchronize processes in a distributed application can be extended to a logical notion of time: the virtual time. In this context we use the concept of a global logical clock, common to all the processes; its progress is defined by rules relevant to the application and not by an external physical phenomenon. The clock allows us to manage the computation, to control its progress, to timestamp events, etc. This paper presents the implementation and evaluation of a distributed kernel to allow the use of virtual time notions in distributed applications. First, the kernel and its interface are presented, and the main implementation choices are discussed. Then the potential uses of this kernel are evaluated and some examples given. Finally, experiment results are presented and analyzed.

Keywords: distributed systems, synchronization, virtual time, distributed simulation, performance evaluation.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE (URA 227) UNIVERSITE DE RENNES I INSA DE RENNES
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE (UNITE DE RECHERCHE DE RENNES)

Table des matières

1 Introduction

Le noyau *Floria* a pour objectif de fournir un ensemble de primitives permettant d'utiliser la notion de temps virtuel dans une application, c'est à dire la possibilité d'ordonner des événements, ou de mesurer des durées, à partir d'une base de temps propre à l'application répartie. La mise en œuvre de ce noyau sur un multiprocesseur à mémoire distribuée a été décrite en détail dans [7]. Le présent rapport a pour objectif de présenter une évaluation détaillée du noyau *Floria* dans le cadre de la simulation répartie, ce type d'application correspondant à l'application principale du concept de temps virtuel.

Dans la section 2 on rappelle les principaux choix de conception et de mise en œuvre du noyau *Floria*: l'utilisation d'un algorithme conservatif avec prévention de l'interblocage [12], l'existence sur chaque processeur de plusieurs processus devant se synchroniser et une mise en œuvre des synchronisations limitant au maximum les blocages.

Notre objectif était de fournir des primitives générales permettant de traiter des problèmes divers; dans la section 3 nous décrivons, à titre d'exemples, quelques utilisations de ces primitives pour modéliser des serveurs classiques d'un réseau de files d'attente.

Enfin, dans la section 4, nous présentons une évaluation quantitative de notre noyau, visant à mesurer les gains obtenus par la parallélisation, étudier l'impact de diverses caractéristiques du modèle sur les performances du simulateur, ainsi que les conséquences sur ces performances des choix de mise en œuvre effectués.

2 Le noyau de gestion du temps virtuel *Floria*

2.1 L'interface de *Floria*

Les applications s'exécutant sur *Floria* sont constituées d'un ensemble de processus, reliés par des canaux de communication *FIFO* selon une topologie quelconque; le seul outil de communication entre ces processus est l'échange de messages. Ces processus évoluent dans un temps virtuel respectant les deux propriétés suivantes :

- $\mathbf{P_1}$: Tous les processus ont la même perception du temps virtuel.
- $\mathbf{P_2}$: Dans le temps virtuel, un message émis à la date t est reçu à la date t.

Un processus P_i peut :

- Consulter l'horloge H_v donnant la date courante dans le temps virtuel.
- Accéder à tous les messages envoyés vers lui à une date inférieure ou égale à la date courante. Ces messages sont stockés dans une file notée $Fconnus_i$, et *Floria* offre des primitives permettant au processus de parcourir cette file et d'y prélever des messages dans l'ordre qu'il souhaite.
- Attendre que l'horloge H_v ait atteint une valeur particulière.
- Attendre qu'un nouveau message lui soit délivré (déposé dans $Fconnus_i$).
- Effectuer des calculs sur des variables locales, ces calculs n'ayant pas de durée dans le temps virtuel.

Seules les primitives d'attente, sur une valeur de l'horloge ou sur l'arrivée d'un message, ont une durée non nulle dans le temps virtuel.

L'utilisateur décrit son application en donnant d'une part le corps des différents types de processus la constituant, et d'autre part en spécifiant le réseau correspondant, c'est à dire le nombre de processus de chaque type à créer, et les canaux reliant ces processus. L'annexe A montre un exemple de description d'un modèle avec les primitives de *Floria*.

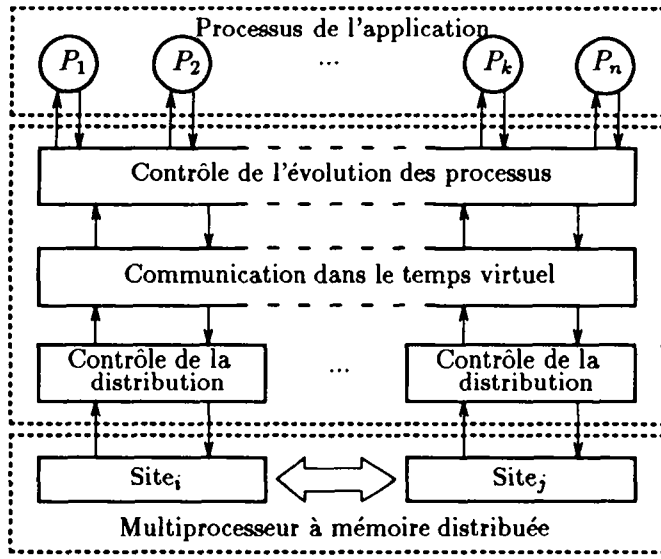

Figure 1 : Architecture générale de *Floria*

2.2 La mise en œuvre du noyau *Floria*

Le noyau *Floria* doit donc permettre d'exécuter une telle application sur un multiprocesseur à mémoire distribuée. Cette exécution impose de résoudre trois problèmes, correspondant à trois niveaux dans l'architecture du noyau (figure 1) :

- Distribuer le réseau, quelconque, de l'application sur une machine ayant un nombre donné de processeurs (sites) et une topologie fixée.
- Réaliser la communication dans le temps virtuel, c'est à dire assurer la propriété $\mathbf{P_2}$ du temps virtuel.
- Contrôler l'évolution des processus.

2.2.1 La distribution de l'application

Nous n'avons pas travaillé sur le placement des processus, qui est fait statiquement. Le rôle de cette partie de *Floria* consiste donc juste à fournir des primitives d'échange de messages asynchrones qui cachent la répartition des processus sur les processeurs : un processus peut ainsi émettre un message sur un de ses canaux de sortie sans se préoccuper de savoir si le destinataire est, ou non, sur le même site.

2.2.2 La mise en œuvre de la communication dans le temps virtuel

Il s'agit de faire en sorte qu'un message émis à la date t dans le temps virtuel soit reçu à la date t. La première propriété du temps virtuel impose que tous les processus aient la même perception du temps virtuel. Comme on travaille sur un multiprocesseur à mémoire distribuée, chaque processus a sa représentation locale de l'horloge virtuelle du système. La synchronisation totale

de toutes ces horloges serait très coûteuse et inutile, en effet comme les processus n'interagissent que par messages, il suffit d'assurer que lors des communications (échanges de messages) les processus ne puissent pas constater de divergences entre leurs horloges locales. En particulier il ne faut pas qu'un processus ait connaissance d'un message émis à t_2 avant d'avoir connaissance d'un message émis à t_1, si $t_2 > t_1$.

Ceci est le problème principal de la simulation distribuée, et deux grandes classes de solution ont été proposées. D'une part les solutions optimistes, dans lesquelles les processus s'assurent a posteriori que les messages sont traités dans l'ordre correct: les messages sont utilisés dans l'ordre de leur arrivée dans le temps réel et un retour en arrière est effectué si on détecte une anomalie dans l'évolution du temps virtuel [9]. D'autre part les solutions pessimistes, ou conservatives, dans lesquelles un algorithme de contrôle s'assure, avant de délivrer un message au processus, que la règle d'évolution dans le temps virtuel est respectée. Ces solutions conduisent à un risque d'interblocage qui doit être soit prévenu [2], soit guéri [1].

Nous avons choisi d'utiliser une méthode conservative avec prévention de l'interblocage. Au moment où nous avons fait ce choix, les méthodes optimistes (*Time Warp*) avaient fait l'objet de plusieurs réalisations et d'évaluations poussées, avec des résultats positifs [10]. Par contre, les méthodes pessimistes avaient été peu évaluées; parmi les quelques travaux sur ce sujet on peut citer [5, 6, 4] où sont décrites des expériences sur des mises en œuvre d'algorithmes conservatifs sur un multiprocesseur à mémoire commune, et [11] qui présente un système complet de simulation répartie basé sur des méthodes pessimistes. C'est pourquoi il nous a semblé intéressant d'investiguer cette voie en réalisant un système complet. De plus, l'efficacité des méthodes optimistes est très fortement liée à celle du retour arrière, c'est à dire en fin de compte à des mécanismes de gestion de la mémoire, alors que nous nous intéressons plus aux algorithmes de contrôle et de synchronisation. Enfin, si des solutions efficaces pour le retour arrière ont été proposées pour des applications particulières comme les files d'attente [3], le retour arrière nous semble plus difficilement généralisable.

L'algorithme de contrôle utilisé repose sur les mécanismes suivants :

- Tous les messages sont estampillés avec leur date d'émission dans le temps virtuel.
- Le *temps-canal* d'un canal d'entrée d'un processus est l'estampille du dernier message reçu sur ce canal.
- L'algorithme de contrôle ne rend un message accessible à un processus (ie. le dépose dans *Fconnus*) que si ce message est le premier du canal ayant le plus petit temps-canal parmi tous les canaux d'entrée du processus.
- Pour éviter que l'absence de message de la simulation sur un canal ne provoque un interblocage, chaque processus émet des messages de contrôle (messages *NULL*) portant une estampille indiquant une date t' supérieure ou égale à sa date courante. Cette valeur t' signifie qu'il n'émettra pas de message sur le canal avant la date t'. On réalise de ce fait une prévision sur le futur du canal, et la valeur de cette prévision constitue un paramètre important pour l'efficacité de la méthode [4].

Comme on a a priori plus de processus que de processeurs, plusieurs processus sont situés sur le même site. Dans ce cas une solution possible est de considérer qu'il existe sur chaque site un simulateur séquentiel classique qui règle l'évolution des processus du site, il n'y a alors qu'une seule horloge virtuelle par site et l'algorithme de contrôle ci-dessus ne s'applique qu'entre sites. Une autre solution consiste à maintenir une horloge par processus, et à appliquer l'algorithme de contrôle entre tous les processus, qu'ils soient ou non situés sur le même site. Nous avons retenu cette deuxième solution bien qu'elle semble a priori plus coûteuse. Nous pensons qu'elle permet de mieux tirer parti du parallélisme potentiel du modèle: il se peut que sur un site, compte tenu du graphe des processus et des interactions possibles avec l'extérieur, on puisse traiter un message qui n'a pas l'estampille la plus petite parmi tous les messages en attente sur le site.

L'algorithme de contrôle de la communication dans le temps virtuel, pour un processus P_i, assure que la file $Fconnus_i$ contient tous les messages émis vers P_i jusqu'à une certaine date He_i.

2.2.3 Le contrôle de l'évolution des processus

Chaque processus P_i est doté d'une représentation locale Hp_i de l'horloge globale Hv. En principe on doit toujours avoir $Hp_i \leq He_i$ (le processus connaît tous les messages qui lui sont destinés jusqu'à l'instant présent). En fait on peut tolérer d'avoir $Hp_i > He_i$ (certains messages émis dans le passé de P_i lui sont inconnus) si le processus n'utilise pas les messages manquants. Le blocage du processus ne devient nécessaire que s'il tente de les utiliser. C'est la solution retenue pour la mise en œuvre des attentes [7]:

- Attente sur horloge: $Attendre(Hv = t)$ se traduit simplement en affectant à l'horloge locale la valeur t, ce qui peut conduire à la situation $Hp_i > He_i$. Il n'y a pas de blocage du processus. Le blocage interviendra éventuellement plus tard, lors de l'accès à la file des messages connus.

- Attente sur message : le processus ne poursuit son exécution que quand un nouveau message a pu être intégré à la file des messages connus.

Donc un processus voit toujours son exécution suspendue parce qu'il attend qu'un nouveau message soit déposé dans sa file *Fconnus*. Un message ne pourra être déposé dans cette file par l'algorithme de contrôle des communications que si celui-ci reçoit un message (*NULL* ou de la simulation) sur le canal d'entrée du processus ayant le plus petit temps-canal. Le numéro de ce canal (*canal bloquant*) est conservé par l'allocateur, le processus en question ne redeviendra activable que quand un message sera effectivement arrivé sur ce canal bloquant. Mais une fois réactivé le processus ne pourra pas forcément poursuivre son évolution : il se peut que le message reçu n'ait pas fait suffisamment progresser l'horloge du processus, dans ce cas le processus se rebloquera immédiatement.

3 Utilisation du noyau *Floria*

Le noyau *Floria* n'est pas le support d'exécution d'un langage particulier, ni la mise en œuvre d'un type particulier de modèle de simulation. Notre objectif est de fournir un ensemble de primitives suffisamment générales pour supporter la réalisation d'environnements correspondant à des langages ou des types de simulation différents. C'est en ce sens que nous parlons de noyau.

Cet objectif est-il atteint? La réponse à cette question, parce que qualitative, est difficile à donner. Nous avons étudié la mise en œuvre d'un certain nombre de modèles classiques, ce qui laisse penser que les primitives de *Floria* permettent effectivement une mise en œuvre simple et raisonnablement efficace de modèles variés. Nous en donnons ici quelques exemples.

Les exemples qui suivent modélisent des serveurs de files d'attente. Dans ce cadre, un serveur est modélisé par un processus et chaque client par un message. Les primitives du noyau *Floria* utilisées dans ces modélisations sont:

- **Vide**: teste si la file *Fconnus* du processus est vide;

- **Premier**: rend une référence (pointeur) sur le premier message disponible dans la file *Fconnus*;

- **Dernier**: idem, pour le dernier message;

- **Prendre** (m): enleve de la file le message désigné par m; une référence sur ce message est rendue;
- **AttendreMsg** (t): attend qu'un nouveau message soit déposé dans la file d'entrée ou que t unités de temps de simulation soient écoulées; une référence sur le nouveau message est rendue (ou NIL, le cas échéant);
- **AttendreTemps** (t): attend l'écoulement de t unités de temps de simulation;
- **Envoyer** (m, c_s): envoie un message m sur un canal de sortie c_s;
- **Prévision** (t): informe le noyau de l'instant minimum du prochain envoi de message;
- **Canal** (m): rend l'identité du canal sur lequel le message m est arrivé.

3.1 mise en œuvre de serveurs *FIFO* et *LIFO*

Un serveur *FIFO* prend chaque client dans l'ordre d'arrivée, traite sa requête pendant un certain temps et l'envoie à la sortie, pour ensuite chercher un autre client. La durée de traitement de chaque client dans cette modélisation est donné par $F_{aléat}$, fonction aléatoire, indépendante du client à traiter.

```
Serveur FIFO:
  répéter
    /* temps de service pour le prochain client et prévision */
    t_serv := F_aléat ;
    Prévision (t_serv) ;

    /* recevoir prochain client à traiter */
    si Vide alors AttendreMsg (∞) fsi ;
    Client := Prendre (Premier) ;

    /* traiter le client et l'expédier */
    AttendreTemps (t_serv) ;
    Envoyer (Client, Sortie) ;
  jusqu'à FinSimulation ;
fin
```

Pour la modélisation du serveur *LIFO*, qui traite toujours le dernier client disponible, il suffit de remplacer l'appel de la primitive **Premier** par **Dernier**, qui rend le dernier message arrivé à l'entrée du processus.

3.2 mise en œuvre de serveurs *QUANTUM*

Dans un serveur *QUANTUM* les clients sont servis à tour de rôle par tranches de temps de durée fixe (appelées *quantum*), l'ordre des services étant de type *FIFO*.

Une solution directe pour la mise en œuvre de ce serveur consisterait à réécrire tout le mécanisme de gestion du serveur à l'intérieur d'un processus. Pourtant une solution beaucoup plus simple peut être appliquée: quand le serveur finit le *quantum* d'un client, celui-ci peut être renvoyé à l'entrée du serveur en utilisant un canal auxiliaire (canal *Retour* – fig. 2). Si après son *quantum* le client requiert encore du temps de traitement, il est renvoyé sur le canal *Retour*, où

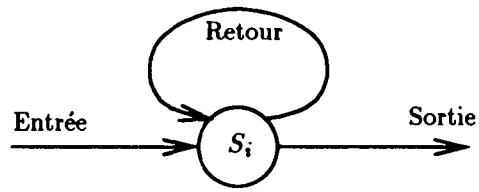

Figure 2 : Serveur *QUANTUM*

il sera mis en ordre avec les nouveaux clients arrivés sur le canal *Entrée*. Une fois terminé son traitement, il abandonne le serveur par le canal *Sortie*.

Le code décrit ci-dessous représente le comportement du serveur. La variable *Retournés*, qui compte le nombre de clients sur le canal *Retour*, est utilisée pour le mécanisme de calcul des prévisions à fournir au noyau. Comme le serveur ne se bloque que par l'absence de clients (donc en particulier quand le canal *Retour* est vide), il n'est pas utile de calculer une bonne prévision tant qu'il y aura des clients sur ce canal (*Retournés* $\neq 0$). Quand le canal *Retour* est vide la prévision est donnée par le minimum entre le prochain temps de service (t_{proch}) et la durée du *quantum* (t_{quant}).

L'attribut t_{rest} de chaque client indique le temps de traitement qui lui reste à effectuer.

```
Serveur QUANTUM:
   t_proch := F_aléat ; /* temps de service pour le premier client */
   Retournés := 0 ;

   répéter
      /* établir une prévision */
      si Retournées = 0 alors Prévision (min (t_proch , t_quant))
      sinon Prévision (0) fsi ;

      /* recevoir prochain client à traiter */
      si Vide alors AttendreMsg (∞) fsi ;
      Client := Prendre (Premier) ;

      /* attribuer le temps de service du nouveau client */
      si Canal (Client) = Retour alors Retournés := Retournés - 1
      sinon
         Client.t_rest := t_proch ;
         t_proch := F_aléat ;
      fsi ;

      /* traiter et expédier le client reçu */
      t_serv := min (Client.t_rest, t_quant) ;
      AttendreTemps (t_serv) ;
      Client.t_rest := Client.t_rest - t_serv ;
      si Client.t_rest = 0 alors Envoyer (Client, Sortie)
      sinon
         Envoyer (Client, Retour) ;
         Retournés := Retournés + 1 ;
      fsi ;
   jusqu'à FinSimulation ;
fin
```

4 Evaluation des performances

Sur un plan quantitatif trois questions se posent :

- D'abord, et c'est le point essentiel, la technique employée permet-elle de gagner un temps d'exécution appréciable en distribuant les simulations ?
- Quel est l'impact des caractéristiques du modèle de simulation sur les performances ? Pour quels types de modèle le noyau se comporte-t-il le mieux ?
- Quel est l'impact, sur les performances du noyau, des choix de mise en œuvre effectués?

Dans la suite de la section nous décrivons d'abord le cadre dans lequel nous avons effectué nos expériences, puis les résultats obtenus. Le résultat d'une expérience peut a priori dépendre d'un grand nombre de facteurs, et il n'est pas facile dans ce cas d'en tirer des leçons. Nous n'avons pas choisi des modèles réels mais plutôt des modèles schématiques réguliers pour pouvoir mieux contrôler les facteurs intervenant dans chaque expérience.

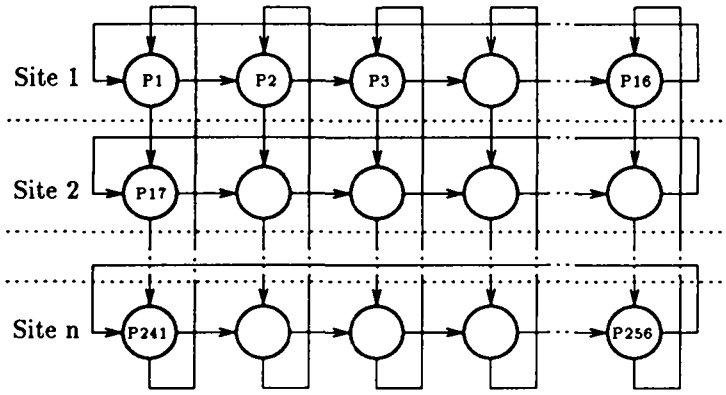

Figure 3 : Schéma du tore

4.1 Le cadre de l'étude

4.1.1 Le support matériel

La mise en œuvre de *Floria* et les expériences ont été effectuées sur l'*iPSC/2* d'*INTEL*, un multiprocesseur à mémoire distribuée avec un réseau de connexion de type hypercube. Chaque processeur *INTEL* 80386 est doté de 4Mo de mémoire locale. Le système d'exploitation sur les nœuds, une version réduite d'*UNIX*, gère les communications entre les sites de façon à fournir une machine virtuelle complètement connectée.

Nous disposons d'une machine ayant 32 processeurs. Il est possible pour un utilisateur de n'utiliser qu'un sous-ensemble de ces processeurs, mais dans ce cas les processeurs alloués ne sont pas partagés avec d'autres utilisateurs. Ceci nous a permis de faire varier le nombre de processeurs sur lesquels nous exécutions nos expériences.

Le système de communication de l'*iPSC/2* assure un contrôle de flux sur les petits messages (moins de 100 octets), ce qui est le cas dans nos expériences. Le nombre de liens de communication séparant deux nœuds a peu d'influence sur le temps de transfert d'un message d'un nœud à l'autre.

4.1.2 Les topologies des modèles simulés

Nous avons étudié principalement deux réseaux: d'une part un tore de 256 processus (16x16), (figure 3) dans lequel chaque processus à deux canaux d'entrée et deux canaux de sortie, et d'autre part un réseau à nombre variable d'entrées noté *rev* (figure 4), constitué d'une matrice 16x16 de processus, dans laquelle on peut faire varier le nombre de canaux d'entrée (et donc de sortie) de chaque processus (dans un réseau à k entrées, noté rev_k, le processus $P_{i,j}$ émet vers les processus $P_{i+1,j+1}$, $P_{i+2,j+1}$, ... $P_{i+k,j+1}$, tous les indices étant calculés modulo 16).

Nous avons choisi des réseaux réguliers pour simplifier l'analyse des résultats des expériences. En particulier l'utilisation de réseaux symétriques permet de limiter les effets du placement des processus.

Ces réseaux comportent un nombre assez important de processus (256). Il nous semble que c'est effectivement le type de modèle à étudier, les simulations intéressantes à distribuer étant forcément grosses.

Les processus sont placés sur les processeurs par ligne, le nombre de lignes placées sur un site dépendant du nombre de processeurs utilisés. Pour éviter les effets liés au placement une ligne

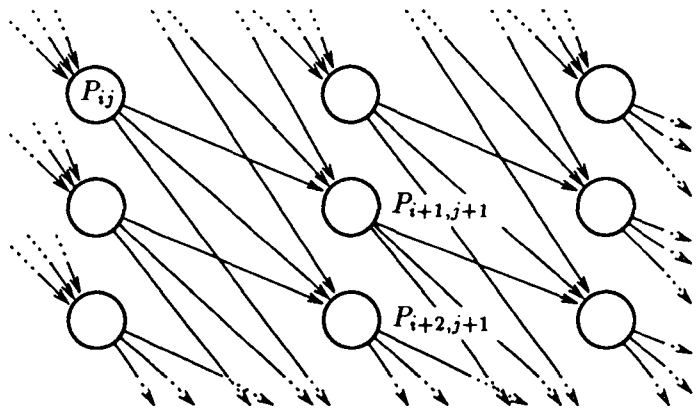

Figure 4 : Schéma du réseau rev_3

n'est jamais éclatée sur deux processeurs, c'est pourquoi nous n'avons pas fait d'expériences avec plus de 16 processeurs.

4.1.3 Le comportement des processus

Dans une expérience tous les processus du modèle ont un comportement semblable, conforme au schéma ci dessous :

```
nbmes   := nombre de messages initiaux;
charge  := charge de calcul physique;
t_sim   := date de fin de simulation;

pour i := 1 jusqu'à nbmess faire /* envoyer les nbmess messages initiaux */
   Msg :=nouveau msg;
   Envoyer (Msg, Sortie);
fait;

tant que Hp < t_sim faire
   /* fixer le prochain temps de service et la prévision */
   t_serv := temps de service pour le msg suivant;
   Prévision ( ... );
   /* prendre le message suivant en fonction de la politique choisie */
   Msg := Prendre ( Premier | Dernier | ... );
   AttendreTemps (t_serv); /* attendre t_serv unités de temps virtuel */
   pour i := 1 jusqu'à charge faire /* charge physique de calcul */
      calcul physique élémentaire;
   fait;
   Envoyer (Msg, Sortie); /* envoyer le message vers la sortie */
fait;
```

Le nombre de messages initiaux, qui fixe le nombre de messages circulant dans le système puisqu'il n'y a pas création de nouveaux messages, est un paramètre des expériences.

Le temps de service correspond à la durée dans le temps virtuel du traitement d'un message, il est donné par un tirage d'une variable aléatoire de loi $(1 + exp\ (9.0))$[1]. Pour chaque processus le germe est différent, et la suite aléatoire obtenue diffère donc d'un processus à l'autre, ce qui évite des synchronisations trop fortes et peu réalistes entre processus.

Nous avons étudié deux politiques pour fixer l'ordre de traitement des messages, *FIFO* (on traite d'abord le plus ancien message reçu) et *LIFO* (on traite d'abord le plus récent message reçu). Nous nous sommes limités à ces deux types de service parce qu'ils représentent deux situations extrêmes pour notre noyau : *FIFO* est la plus favorable (le prélèvement d'un message ne provoque un blocage que s'il n'y a pas de message), *LIFO* est la plus défavorable (il faut toujours attendre que $HP_i <= He_i$ pour prélever un message).

Le paramètre *charge de calcul* permet de faire varier le temps physique passé à traiter un message. La durée d'un calcul physique élémentaire est de l'ordre de 0.6 millisecondes.

La prévision mesure la capacité d'un processus à connaître le futur de ses canaux de sortie. Si au temps virtuel t un processus fixe la valeur de sa prévision sur un canal de sortie à une valeur δ, cela signifie qu'il assure ne pas émettre de message de la simulation sur ce canal avant $t + \delta$. Dans nos expériences la prévision est toujours la même sur tous les canaux de sortie d'un processus. Plus la valeur proposée est proche de la valeur réelle, plus la prévision est de bonne qualité. Cette qualité de la prévision est évaluée par le rapport *prévision / (prochain temps de service)*; quand on prend comme prévision le prochain temps de service on a la qualité maximum (qualité 1), quand on prend comme prévision le temps de service minimum on a la qualité minimum (qualité 0). Pour pouvoir faire évoluer de manière plus fine ce paramètre on a utilisé comme prévision un pourcentage x de la prévision maximum (qualité x).

Les messages sont émis sur les canaux de sortie de manière régulière, un canal après l'autre circulairement.

4.1.4 Stabilité des résultats des expériences

Il y a deux sources de variation sur les résultats d'expériences, pour un ensemble de paramètres donné :

- Les variables aléatoires intervenant dans le modèle : ceci a évidement un effet sur le résultat de la simulation (le comportement du modèle), mais aussi sur le comportement du noyau (l'évolution différente du temps virtuel conduit à des situations de synchronisation différentes).
- Les délais variables de communication dans l'*iPSC/2* : ils peuvent, bien sûr, avoir un effet sur le comportement du noyau, mais ils peuvent également influer sur le comportement de certains modèles. En effet, comme *Floria* permet de délivrer un message émis à t sans que tous les messages émis à t soient reçus, des délais de communications différents peuvent conduire à des ordres d'arrivée des messages différents et donc à des comportements du modèle différents.

Nous avons d'abord étudié la stabilité des résultats des expériences quand tous les paramètres, y compris les germes des séries pseudo-aléatoires, sont identiques. Dans ce cas, les seules variations sont dûes aux délais de communication de l'*iPSC/2*. Les modèles avec des processus *FIFO* sont moins stables que les modèles avec des processus *LIFO* : en effet pour des processus *LIFO* les temps de communication n'ont aucun effet sur le comportement du modèle puisque un message émis à t ne peut être délivré que si tous les messages émis à t ont été reçus. Pourtant, dans tous les cas l'effet moyen sur le comportement du modèle est très faible : l'intervalle de confiance à 95% est inférieur à $\pm 3\%$ de la valeur moyenne, comme l'indique le tableau 1 qui donne ce pourcentage pour le temps de simulation, le nombre de messages de la simulation et le nombre de messages *NULL*.

[1] $exp\ (x)$: distribution exponentielle négative de moyenne x.

réseau	nombre nœuds	type	charge calcul	nombre msgs	prév	confiance temps	confiance msgs simul	confiance msgs *NULL*
tore	4	FIFO	0	1	min	1,61	0,27	1,95
tore	4	LIFO	0	1	min	0,71	0,00	0,66
rev_4	4	FIFO	0	1	min	2,49	0,27	2,60
rev_4	4	FIFO	0	4	min	1,35	0,13	1,39
rev_4	4	LIFO	0	1	min	0,81	0,00	0,74

Table 1 : Largeur de l'intervalle de confiance en % de la moyenne

Nous avons également étudié les résultats des expériences faites avec les mêmes paramètres, mais avec des séries pseudo-aléatoires différentes. Là encore nous avons une bonne stabilité des résultats des expériences, l'intervalle de confiance à 95% est toujours inférieur à $\pm 3\%$ de la valeur moyenne, comme l'indique le tableau 2 pour quelques expériences.

réseau	nombre nœuds	type	charge calcul	nombre msgs	prév	confiance temps	confiance msgs simul	confiance msgs *NULL*
tore	4	FIFO	0	1	min	1,85	0,43	2,50
tore	4	LIFO	0	1	min	1,88	0,27	1,53
rev_4	4	FIFO	0	1	min	2,21	0,37	2,61
rev_4	4	FIFO	0	4	min	1,72	0,29	2,41
rev_4	4	FIFO	4	1	min	2,75	0,32	2,69
rev_4	4	LIFO	0	4	min	1,53	0,23	1,67
rev_4	4	LIFO	0	1	min	1,80	0,27	1,86

Table 2 : Largeur de l'intervalle de confiance en % de la moyenne

4.1.5 Comparaison avec un simulateur séquentiel

Pour évaluer le gain apporté par une simulation distribuée sur n processeurs il faut en principe la comparer avec la simulation du même modèle effectuée sur un simulateur séquentiel. Nous ne disposons pas d'un simulateur séquentiel pouvant s'exécuter sur un nœud de l'*iPSC/2*.

Comme d'une part nous disposons d'un logiciel de traitement de réseaux de file d'attente *Qnap2* [13], incluant un simulateur séquentiel et s'exécutant sur une station de travail, et que d'autre part nous pouvons exécuter *Floria* sur cette station, nous avons voulu calibrer nos expériences en comparant l'exécution de la simulation du même modèle sur une station de travail, en utilisant soit *Qnap2* soit *Floria*.

Le tableau 3 regroupe les résultats de quelques unes de ces expériences sur un réseau *rev* de 256 processus (16x16): la colonne *Msg* donne le nombre de messages par processus, *Can* donne le nombre de canaux d'entrée par processus, *Prév* indique quelle prévision a été utilisée. Les temps de simulation (T_{Qnap2} et T_{Floria}) sont donnés en secondes, et $\%Diff = (T_{Floria} - T_{Qnap2})/T_{Qnap2}$. On constate que sur les réseaux étudiés l'exécution de *Floria* sur un monoprocesseur est la plupart du temps plus rapide que l'exécution de *Qnap2* ($\%Diff$ négatif). Les expériences 20 et 68 correspondent à des situations très pénalisantes pour l'algorithme de contrôle de *Floria*, comme on le verra par la suite (beaucoup de canaux d'entrée, peu de messages et une prévision de qualité minimale).

Exp	Type	Dim	Msg	Can	Prév	T_{Qnap2}	T_{Floria}	% Diff
19	FIFO	16	1	2	Min	108.9	99.0	-9.1
20	FIFO	16	1	4	Min	108.3	687.0	534.3
21	FIFO	16	2	2	Min	164.0	59.0	-64.0
22	FIFO	16	2	4	Min	165.8	185.0	11.6
23	FIFO	16	4	2	Min	214.3	45.0	-79.0
24	FIFO	16	4	4	Min	214.5	87.0	-59.4
43	FIFO	16	1	2	Max	108.9	49.0	-55.0
44	FIFO	16	1	4	Max	108.3	72.0	-33.5
45	FIFO	16	2	2	Max	164.0	42.0	-74.4
46	FIFO	16	2	4	Max	165.8	69.0	-58.4
47	FIFO	16	4	2	Max	214.3	42.0	-80.4
48	FIFO	16	4	4	Max	214.5	57.0	-73.4
67	LIFO	16	1	2	Min	109.1	150.0	37.5
68	LIFO	16	1	4	Min	108.3	635.0	486.3
69	LIFO	16	2	2	Min	163.8	109.0	-33.5
70	LIFO	16	2	4	Min	164.2	219.0	33.4
71	LIFO	16	4	2	Min	212.7	97.0	-54.4
72	LIFO	16	4	4	Min	212.7	173.0	-18.7
91	LIFO	16	1	2	Max	109.1	52.0	-52.3
92	LIFO	16	1	4	Max	108.3	75.0	-30.7
93	LIFO	16	2	2	Max	163.8	51.0	-68.9
94	LIFO	16	2	4	Max	164.2	71.0	-56.8
95	LIFO	16	4	2	Max	212.7	53.0	-75.1
96	LIFO	16	4	4	Max	212.7	82.0	-61.4

Table 3 : Comparaison entre *Qnap2* et *Floria* / modèle à 256 processus

Un certain nombre de facteurs peuvent expliquer cette bonne performance de *Floria*: d'une part *Qnap2* peut faire bien d'autres choses que la simulation séquentielle (résolution analytique par diverses méthodes) et le simulateur n'est pas forcément très optimisé; d'autre part on a utilisé un gros modèle, ce qui est pénalisant pour un simulateur séquentiel. Effectivement la simulation d'un modèle plus petit (16 processus) est plus favorable à *Qnap2*, comme on le voit sur le tableau 4.

Compte tenu de ces résultats, pour les expériences ultérieures nous avons évalué les accélérations en rapportant une simulation avec *Floria* sur n processeurs de l'*iPSC/2* à la simulation du même modèle avec *Floria* sur un seul processeur.

4.2 Résultats des expériences

Nous présentons dans cette section un certain nombre de résultats issus des expériences effectuées. Le tableau 5 donne les caractéristiques détaillées de chacune de ces expériences, nous ne les rappellerons pas à chaque fois.

4.2.1 Accélération obtenue par la distribution

Nous avons calculé l'accélération comme étant, pour un modèle donné, le temps d'exécution de *Floria* sur 1 site divisé par le temps d'exécution sur n sites. La figure 5 donne les résultats obtenus sur le tore 16x16, avec des serveurs *FIFO* et un indice de charge de calcul physique

Exp	Type	Dim	Msg	Can	Prév	T_{Qnap2}	T_{Floria}	% Diff
1	FIFO	4	1	2	Min	4.67	19.0	306.9
2	FIFO	4	1	4	Min	4.59	42.0	815.0
3	FIFO	4	2	2	Min	5.86	8.0	36.5
4	FIFO	4	2	4	Min	5.98	26.0	334.8
5	FIFO	4	4	2	Min	6.64	5.0	-24.7
6	FIFO	4	4	4	Min	6.84	13.0	90.1
25	FIFO	4	1	2	Max	4.67	6.0	28.5
26	FIFO	4	1	4	Max	4.59	11.0	139.7
27	FIFO	4	2	2	Max	5.86	5.0	-14.7
28	FIFO	4	2	4	Max	5.98	8.0	33.8
29	FIFO	4	4	2	Max	6.64	4.0	-39.8
30	FIFO	4	4	4	Max	6.84	5.0	-26.9
49	LIFO	4	1	2	Min	4.39	26.0	492.3
50	LIFO	4	1	4	Min	4.39	44.0	902.3
51	LIFO	4	2	2	Min	5.74	16.0	178.7
52	LIFO	4	2	4	Min	5.93	34.0	473.4
53	LIFO	4	4	2	Min	6.57	13.0	97.9
54	LIFO	4	4	4	Min	6.73	26.0	286.3
73	LIFO	4	1	2	Max	4.39	9.0	105.0
74	LIFO	4	1	4	Max	4.39	12.0	173.3
75	LIFO	4	2	2	Max	5.74	8.0	39.4
76	LIFO	4	2	4	Max	5.93	11.0	85.5
77	LIFO	4	4	2	Max	6.57	7.0	6.5
78	LIFO	4	4	4	Max	6.73	10.0	48.6

Table 4 : Comparaison entre *Qnap2* et *Floria* / modèle à 16 processus

variant de 0 à 16 (expérience 1). On obtient dans tous les cas une accélération de l'exécution quand on augmente le nombre de processeurs, cette accélération étant plus forte quand la charge de calcul physique est plus élevée.

L'efficacité, c'est à dire l'accélération divisée par le nombre de processeurs, diminue avec le nombre de processeurs utilisés, comme le montre la figure 6. Plus on utilise de processeurs, moins ils sont efficacement utilisés. Mais en fait pour la simulation distribuée, destinée à rendre réalisables de très grosses simulations, c'est bien la possibilité de diminuer le temps global de simulation qui est importante, même au prix d'une mauvaise utilisation des processeurs.

Cette possibilité d'accélération se retrouve pour d'autres modèles, comme on le voit par exemple sur les figures 7 (avec les mêmes conditions que précédemment, mais des serveurs *LIFO* - expérience 2) et 8 (*rev* à 2, 3, 4 ou 8 canaux d'entrée - expérience 3).

4.2.2 Influence de la charge de calcul

Nous allons étudier ici plus précisément l'effet de la charge de calcul physique sur le comportement d'une simulation. Pour un indice de charge de calcul physique i, le temps de simulation est donné par $T_{sim} = T_{noyau} + T_{file} + i * T_{calcul}$, avec :

- T_{sim} = temps de simulation;
- T_{noyau} = temps total passé dans le noyau *Floria*;

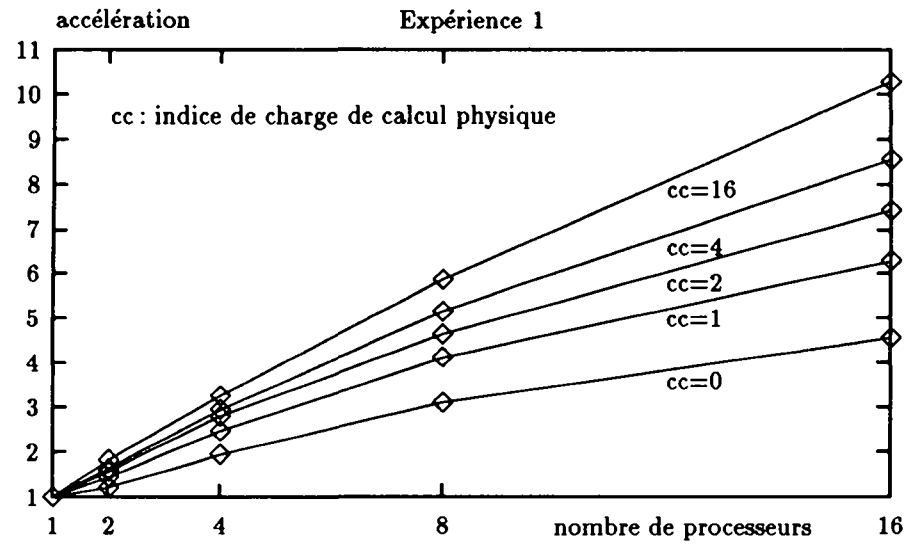

Figure 5 : Accélération = $\mathcal{F}$ (nombre de sites) pour le tore, processus *FIFO*

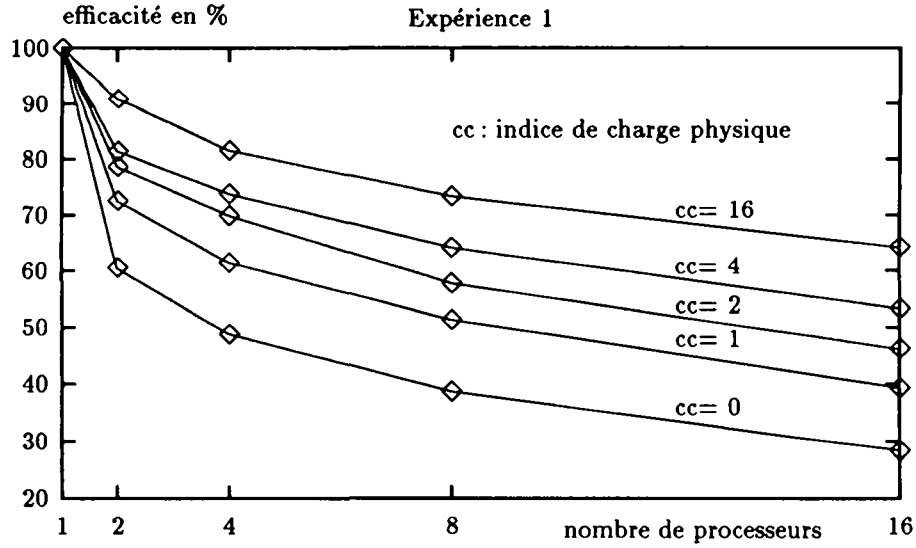

Figure 6 : Efficacité = $\mathcal{F}$ (nombre de sites) pour le tore, processus *FIFO*

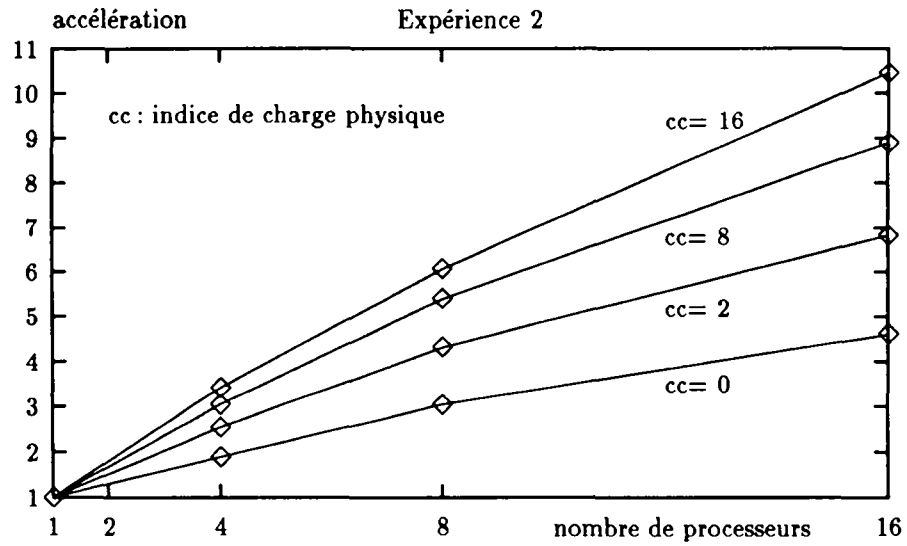

Figure 7 : Accélération = $\mathcal{F}$ (nombre de sites) pour le tore, processus *LIFO*

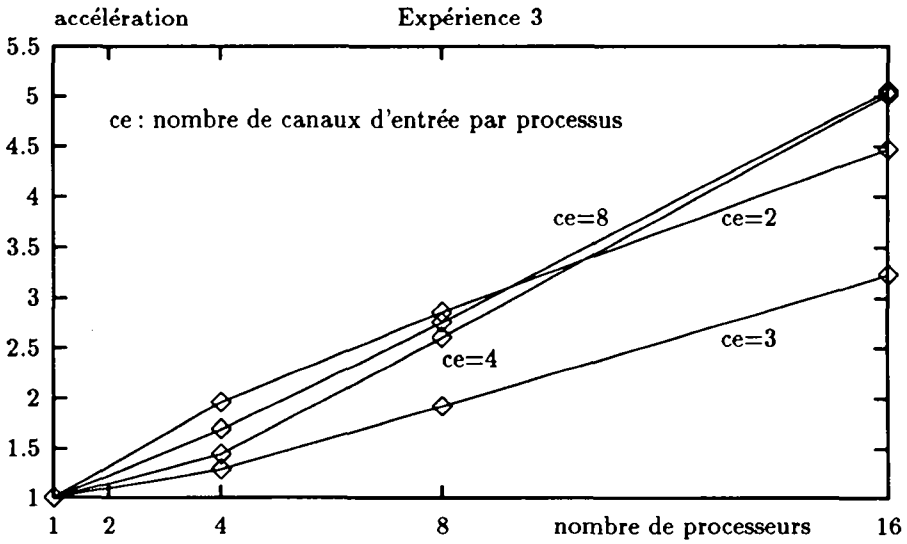

Figure 8 : Accélération = $\mathcal{F}$ (nombre de sites) pour le *rev*, processus *LIFO*

exp	réseau	nombre canaux	nombre nœuds	type processus	charge calcul	nombre msgs	prév
1	tore	2	1 à 16	FIFO	0 à 16	1	0
2	tore	2	1 à 16	LIFO	0 à 16	1	0
3	*rev*	2-3-4-8	1 à 16	FIFO	0	1	0
4	tore	2	4-8-16	FIFO/LIFO	0 à 64	1	0
5	*rev*	4	4-8-16	FIFO	0 à 32	1	0
6	tore	2	8	FIFO/LIFO	0	1	0
7	tore	2	8	FIFO/LIFO	0	1-2-3-4	0
8	tore	2	8	FIFO/LIFO	0	1-2	0 à 1
9	*rev*	2-3-4-5	8	FIFO/LIFO	0	1-2-4	0
10	tore	2	8	FIFO/LIFO	0	1-2	0 à 1
11	*rev*	3-5	8	FIFO	0	1-4	0 à 1
12	*rev*	3-5	8	LIFO	0	1-4	0 à 1
13	tore	2	8	FIFO/LIFO	0	1-2-3-4	0 et 1
14	*rev*	2-3-4-5	8	FIFO	0	1-2-3-4	0
15	*rev*	2-3-4-5	8	LIFO	0	1-2-3-4	0
16	*rev*	1 à 15	16	FIFO/LIFO	0	1	0 et 1

Table 5 : Caractéristiques des expériences

- T_{file} = temps passé dans le modèle pour la mise en œuvre de la politique de gestion de la file;
- T_{calcul} = temps passé dans le modèle à faire un calcul physique élémentaire.

Intuitivement, plus les processus du modèle ont de calcul à faire, plus la distribution de la simulation doit être intéressante. On retrouve effectivement ce résultat, par exemple sur la figure 9 qui donne l'évolution de l'efficacité en fonction de la charge de calcul physique, pour un tore 16x16 (expérience 4).

Si l'augmentation de l'efficacité est très rapide au début, elle se stabilise au delà d'une certaine charge. Dans l'expérience de la figure 9 on peut considérer qu'avec un indice de charge de 16 on a pratiquement atteint l'optimum, ce qui correspond à un temps physique de traitement d'un message de 9 ms. La figure 10 donne pour la même expérience, dans le cas des serveurs *FIFO*, le pourcentage de temps passé dans le noyau par rapport au temps total de simulation. On constate que le pourcentage de temps passé dans le système diminue rapidement quand la charge physique commence à augmenter, pour se stabiliser ensuite. Ce phénomène n'est pas dû à une évolution du nombre de messages *NULL*, qui n'est pas sensible à l'augmentation de la charge, comme on le voit sur la figure 11, donnant le pourcentage de messages *NULL* par rapport au nombre total de messages échangés. La diminution du pourcentage de temps passé dans le système quand la charge commence à augmenter est probablement dûe à la récupération partielle des temps d'attente pour effectuer le travail supplémentaire dû au modèle. Mais cette récupération ne peut être complète, et il arrive un moment ou l'augmentation de la charge n'améliore plus les performances.

On retrouve un comportement similaire avec le réseau à nombre variable d'entrées, comme on le voit sur la figure 12, qui donne l'efficacité en fonction de la charge pour un réseau à 4 canaux d'entrée par processus *FIFO* (expérience 5).

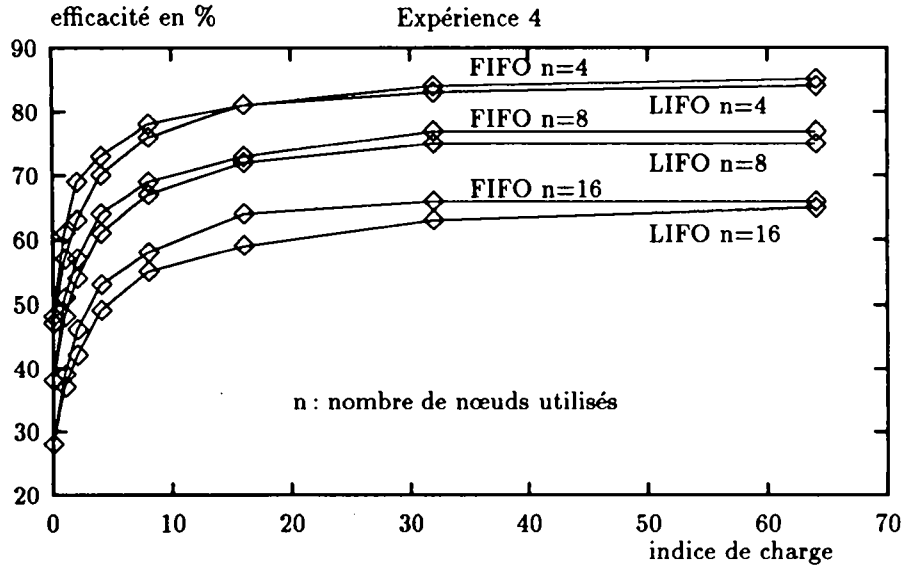

Figure 9 : Efficacité = $\mathcal{F}$ (charge de calcul) pour le tore

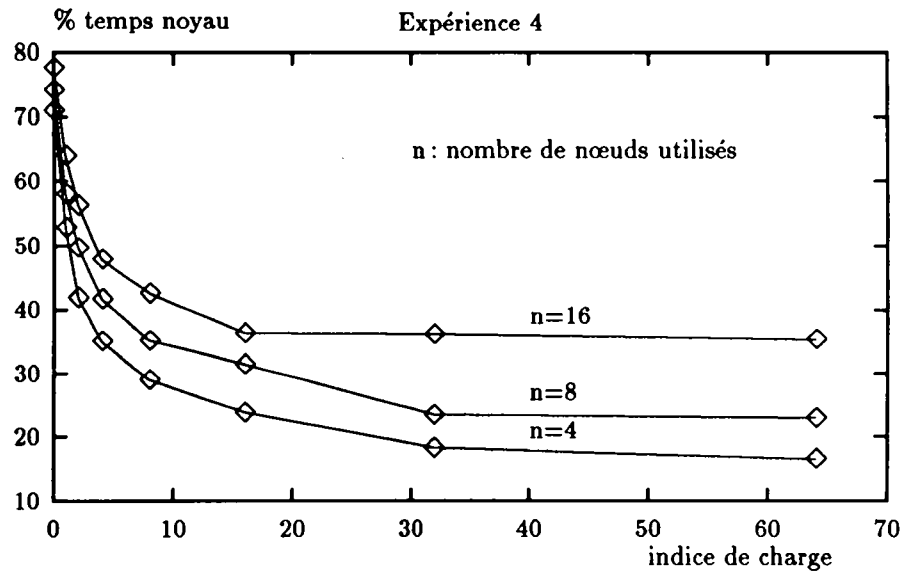

Figure 10 : % temps noyau = $\mathcal{F}$ (charge de calcul) pour le tore, processus *FIFO*

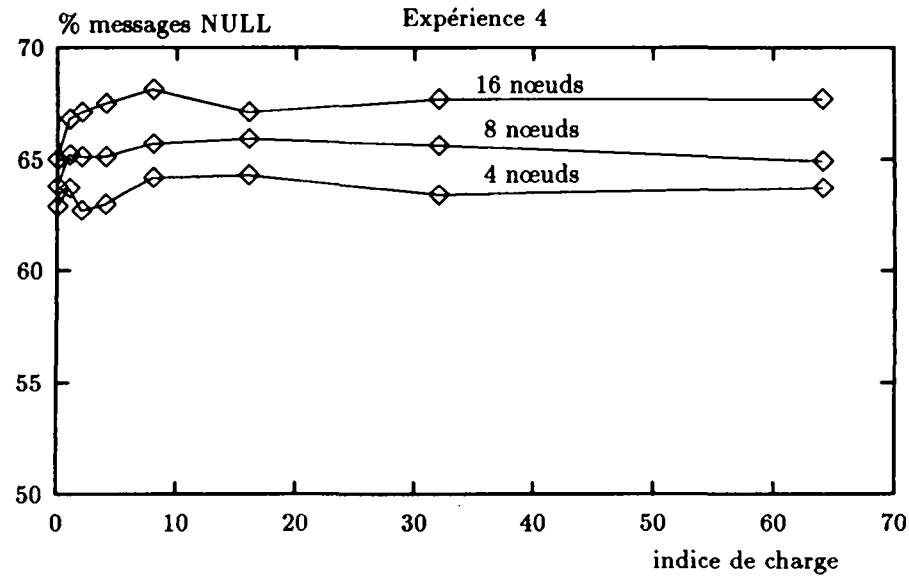

Figure 11 : % messages *NULL* = $\mathcal{F}$ (charge de calcul) pour le tore, processus *FIFO*

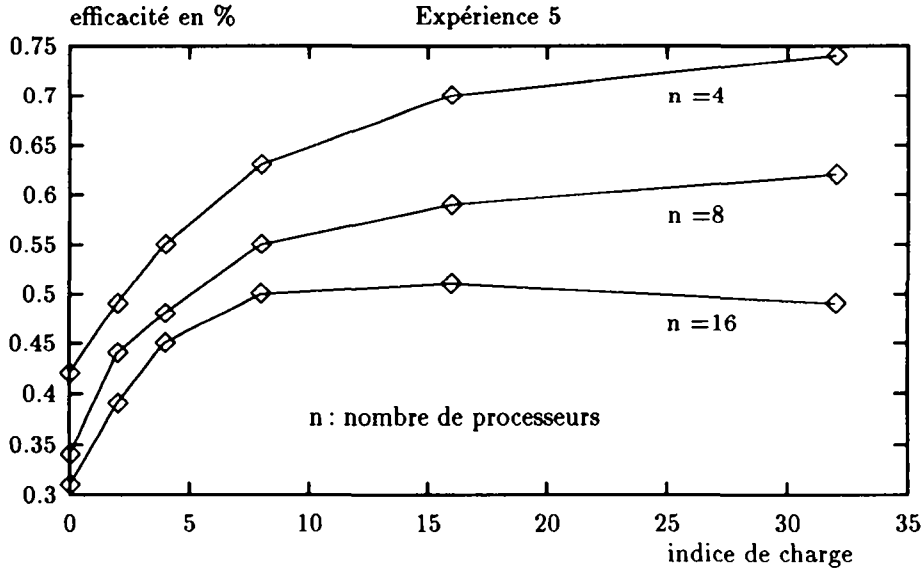

Figure 12 : Efficacité = $\mathcal{F}$ (charge de calcul) pour le *rev* à 4 canaux d'entrée

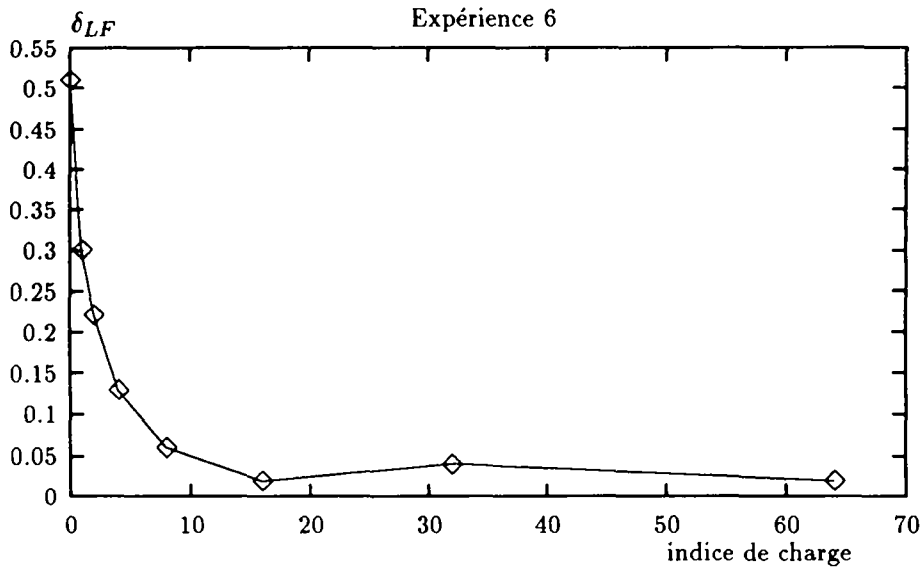

Figure 13 : $\delta_{LF} = \mathcal{F}$ (charge de calcul) pour le tore

4.2.3 Influence de la politique de service

Nous étudions ici l'influence du comportement des processus du modèle simulé sur les performances du noyau. Nous avons étudié deux types de comportement : des serveurs *FIFO* et des serveurs *LIFO*. Dans chaque expérience tous les processus sont du même type. On évalue, tous paramètres étant égaux par ailleurs, la différence entre les temps de simulation pour des serveurs *FIFO* et *LIFO*, grâce à l'indicateur $\delta_{LF} = (T_{LIFO} - T_{FIFO})/T_{FIFO}$.

La courbe 13 donne la valeur de δ_{LF} en fonction de l'indicateur de charge physique, pour un tore sur 8 processeurs (expérience 6). La différence, importante pour les charges faibles, devient négligeable quand la charge augmente. Cela confirme les résultats obtenus sur l'influence de la charge physique : les temps d'attente plus importants pour les serveurs *LIFO* expliquent l'écart important quand la charge est faible, cet écart diminue quand le travail supplémentaire du modèle est pris sur ces temps d'attente.

La courbe 14 donne la valeur de δ_{LF} en fonction du nombre de messages par processus, pour un tore sur 8 processeurs (expérience 7). On constate que l'écart est plus important quand le nombre de messages augmente, cela traduit l'efficacité de notre contrôle de l'exécution des processus : le fait de ne bloquer un processus que s'il tente effectivement d'utiliser un message manquant conduit à diminuer considérablement le nombre de blocages pour des serveurs *FIFO* quand le nombre de messages augmente.

La courbe 15 donne la valeur de δ_{LF} en fonction de la qualité de la prévision (0 correspond à une prévision minimale, et 1 à une prévision maximale), pour un tore sur 8 processeurs (expérience 8). Une bonne prévision diminue l'écart entre le comportement des processus *FIFO* et *LIFO*.

La courbe 16 donne la valeur de δ_{LF} en fonction du nombre de canaux d'entrée pour le *rev* sur 8 processeurs (expérience 9). On retrouve l'augmentation de la différence entre les serveurs *FIFO* et *LIFO* quand le nombre de messages en circulation augmente. Le comportement vis à

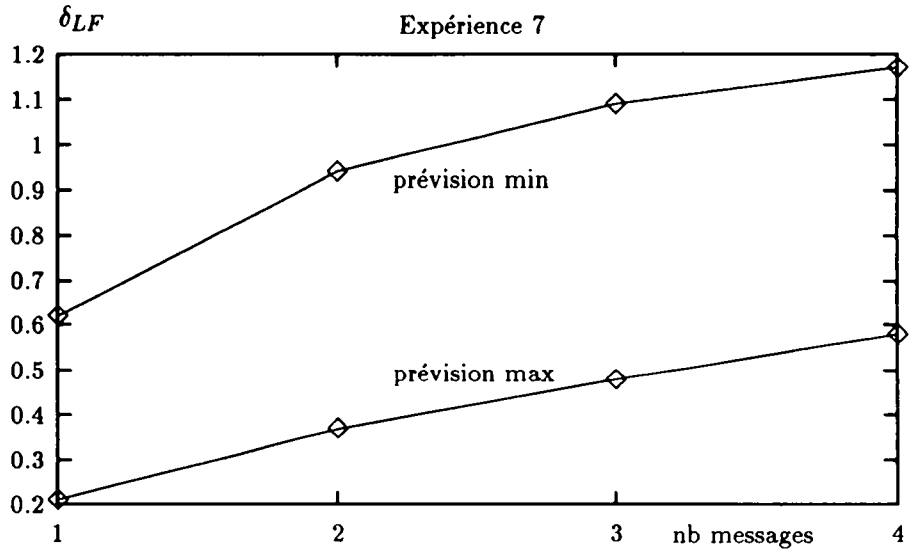

Figure 14 : $\delta_{LF} = \mathcal{F}$ (nombre de messages) pour le tore

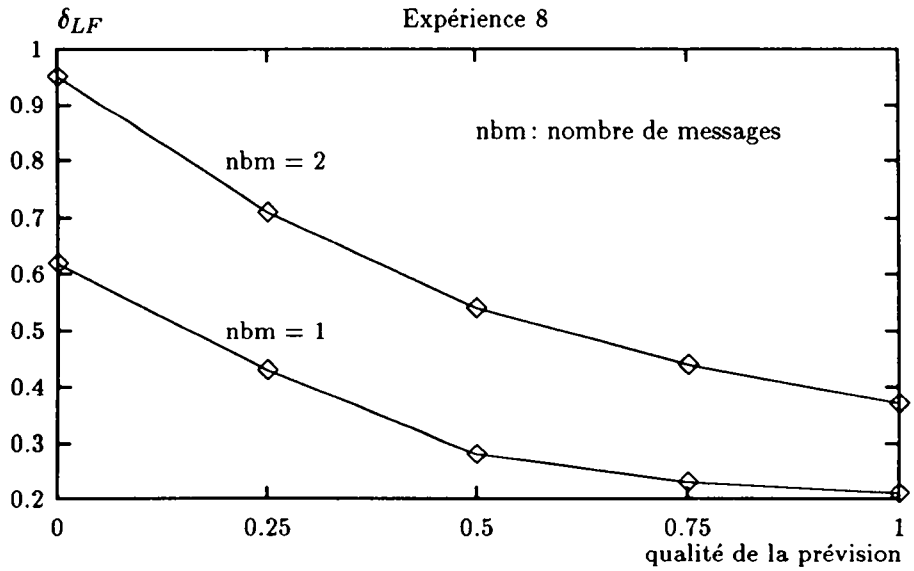

Figure 15 : $\delta_{LF} = \mathcal{F}$ (qualité de la prévision) pour le tore

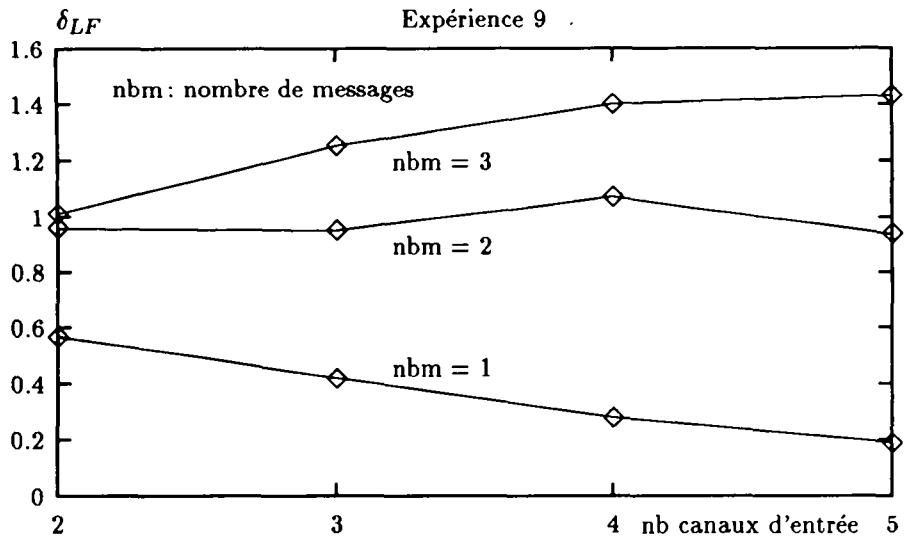

Figure 16 : $\delta_{LF} = \mathcal{F}$ (nombre de canaux d'entrée) pour le *rev*

vis du nombre de canaux d'entrée (NB_{ce}) des processus dépend du nombre de messages : quand il y a un message par processus l'augmentation de NB_{ce} est très pénalisante quel que soit le type de serveur, et la différence entre les deux types de serveurs diminue; par contre quand le nombre de messages augmente le phénomène évoqué précédemment est retrouvé et les serveurs *FIFO* sont avantagés.

4.2.4 Influence de la prévision

On s'intéresse ici plus particulièrement aux effets de la prévision sur le comportement des simulations. Une étude faite par Fujimoto [4] sur les méthodes de simulation distribuées conservatives indiquait que la qualité de la prévision (*lookahead*) était un paramètre influençant fortement l'efficacité de la simulation. Cette étude avait été effectuée sur un multiprocesseur à mémoire partagée (*BBN*) et il est intéressant de regarder si ses résultats restent valables avec une mémoire distribuée. Dans son étude, Fujimoto prenait toujours comme prévision le temps de service minimum, et il évaluait la qualité de la prévision par le rapport *prévision / (temps moyen de service)*. Pour faire varier cette qualité il utilisait plusieurs lois différentes, de même moyenne.

Pour éviter des effets indésirables, nous avons préféré conserver toujours la même loi pour le temps de service. Nous avons estimé la prévision maximale comme étant le prochain temps de service (qualité 1), et nous fournissons au noyau *Floria* une prévision égale à un certain pourcentage x de cette prévision maximale (qualité x). La prévision minimale correspond au temps de service minimum (qualité 0).

La figure 17 donne le temps de simulation en fonction de la qualité de la prévision pour un tore 16x16 (expérience 10). On constate que la qualité de la prévision a plus d'importance pour des serveurs *LIFO* que *FIFO*. C'est une conséquence de l'optimisation du blocage des processus faite dans *Floria*, en particulier s'il y a suffisamment de messages (deux par processus dans ce cas-ci) les processus *FIFO* ne se bloquent pratiquement plus et la prévision n'a plus d'influence.

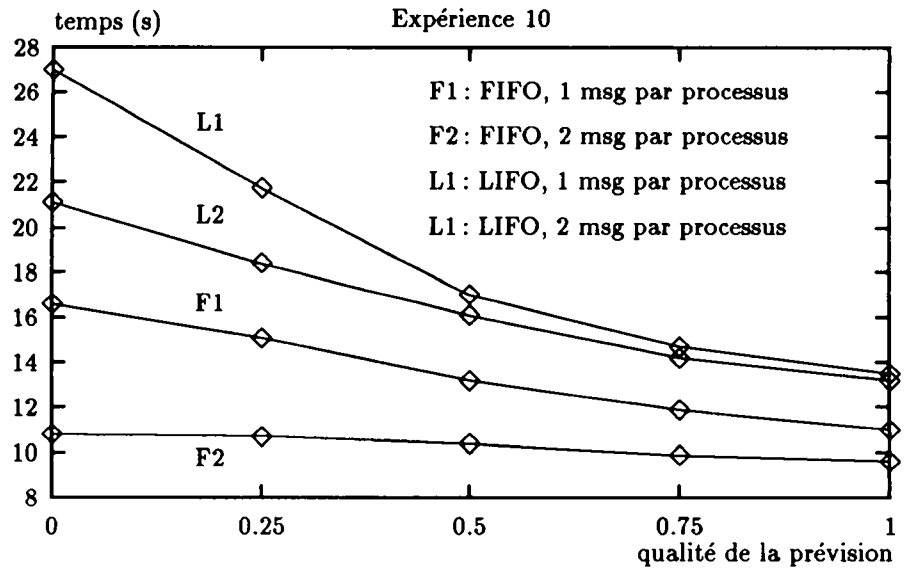

Figure 17 : temps = $\mathcal{F}$ (qualité de la prévision) pour le tore

On retrouve les résultats de Fujimoto pour les processus *LIFO*.

La figure 18 donne, pour la même expérience, le pourcentage de messages *NULL* émis par rapport au nombre total de messages. Si l'augmentation de la qualité de la prévision permet de diminuer le nombre de messages *NULL* pour des serveurs *LIFO*, par contre ce paramètre à peu d'influence sur le nombre de messages *NULL* pour des serveurs *FIFO*. Pour des serveurs *LIFO* on se sert effectivement des messages *NULL* pour pouvoir faire progresser l'horloge d'entrée He, et il peut être nécessaire de consommer plusieurs messages *NULL* avant de pouvoir délivrer au processus un message de la simulation ($\%NULL > 50\%$). Par contre, pour des processus *FIFO*, compte tenu de la mise en œuvre des attentes, il y a souvent un message disponible pour le processus sans qu'il doive se bloquer, et on émet, en moyenne, au plus un message *NULL* par message de la simulation.

Les figures 19 et 20 donnent le temps de simulation en fonction de la qualité de la prévision sur le réseau à nombre variable d'entrées (expérience 11). La qualité de la prévision est un facteur important, même pour des processus *FIFO*, quand le nombre de canaux d'entrée par processus est élevé. Cet effet tend à diminuer quand le nombre de messages augmente.

4.2.5 Influence du nombre de messages

Nous étudions ici l'effet du nombre de messages en circulation sur le comportement du noyau. Le nombre de messages reste constant pendant une simulation, la charge en messages est fixée par le nombre de messages émis initialement par chaque processus; le traitement d'un message produit toujours l'émission d'un message de la simulation. Les expériences ont été faites sur une durée de simulation fixée (5000 unités de temps virtuel), avec la même loi pour les temps de services; l'augmentation du nombre de messages en circulation ne conduit donc pas à une augmentation du nombre de messages traités pendant une expérience.

L'augmentation du nombre de messages est pénalisante pour un simulateur séquentiel à cause

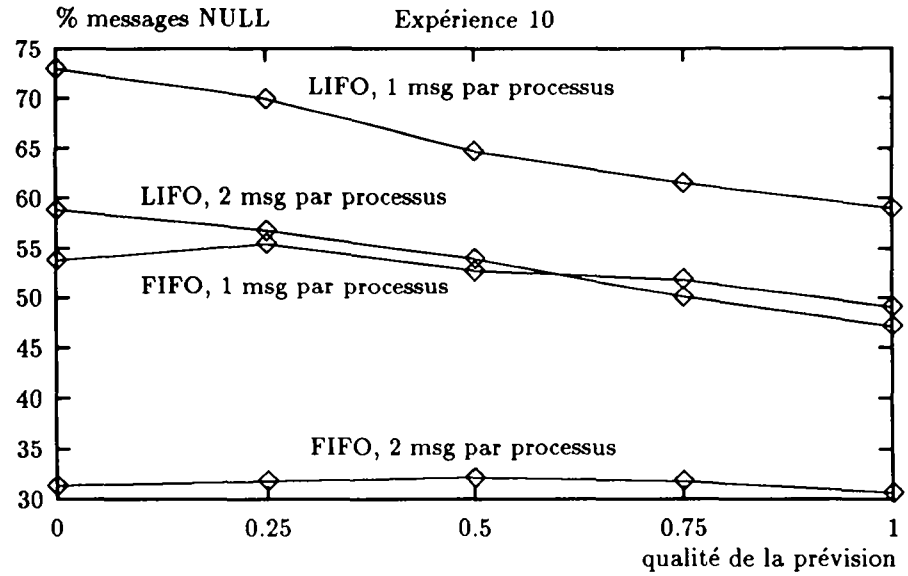

Figure 18 : % $NULL = \mathcal{F}$ (qualité de la prévision) pour le tore

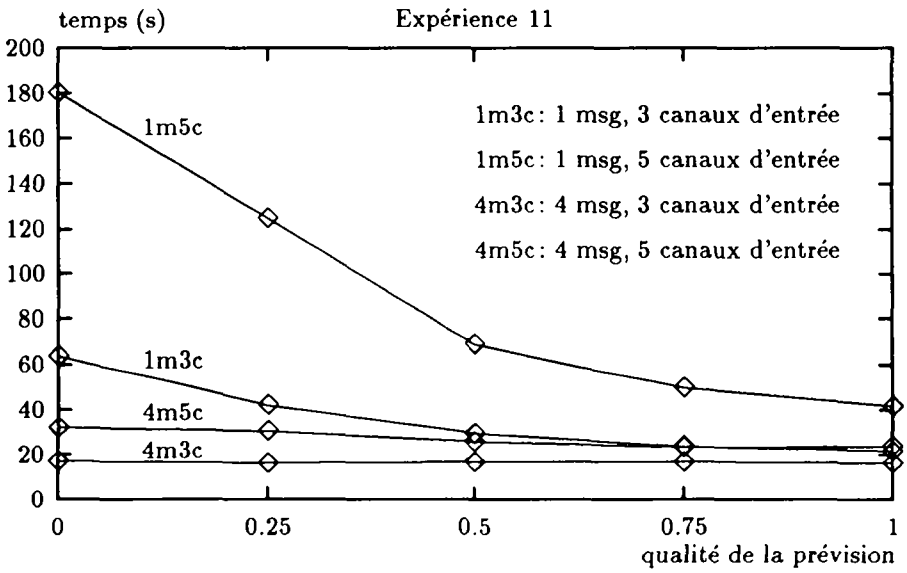

Figure 19 : temps = $\mathcal{F}$ (qualité de la prévision) pour le *rev*, processus *FIFO*

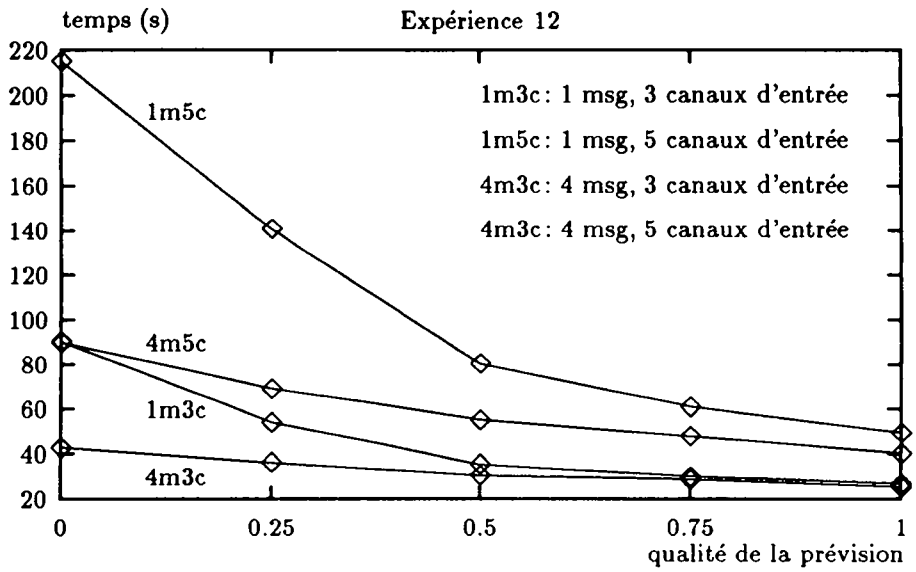

Figure 20 : temps = $\mathcal{F}$ (qualité de la prévision) pour le *rev*, processus *LIFO*

de l'augmentation de la longueur de la file des événements. Dans une simulation distribuée la taille des files de messages augmente également, mais ce phénomène est moins important parce qu'on a une file par processus, et non pas une file unique. De plus cet effet se fait surtout sentir pour des processus *LIFO* (il faut parcourir toute la file pour accéder au dernier message). Par contre, intuitivement, comme l'algorithme de contrôle de la communication attend d'avoir un message sur chaque canal d'entrée d'un processus pour lui délivrer un message, l'augmentation du nombre de messages devrait conduire à une amélioration des performances de la simulation, grâce à une diminution des attentes.

La figure 21, donnant l'accélération pour une simulation du tore sur 4 processeurs, confirme partiellement cette intuition (expérience 13). L'amélioration se produit effectivement pour des processus *FIFO*, par contre il n'y a pratiquement plus d'amélioration pour des processus *LIFO* au delà de deux messages. La figure 22 donne, pour les mêmes expériences, le temps de simulation en fonction du nombre de messages par processus. On constate que le nombre de messages a peu d'influence sur la durée totale de simulation quand on utilise une prévision maximale, l'augmentation de l'accélération dans ces cas est donc dûe essentiellement à l'augmentation du temps de simulation sur un seul processeur quand la charge en messages augmente. Par contre avec une prévision minimale le fait de passer de un à deux messages par processus fait diminuer le temps de simulation sur 4 processeurs : l'augmentation du nombre de messages de la simulation fait diminuer la nécessité d'attendre des messages *NULL*. Au delà de deux messages par processus, il n'y a plus d'effet. Cela est dû à la topologie du tore : un processus a deux canaux d'entrée, quand aucun canal d'entrée n'est vide (en moyenne), les messages supplémentaires n'améliorent plus les performances.

La figure 23 donne, pour les mêmes expériences, le temps passé dans l'exécution du modèle par opposition au temps passé dans le noyau, ce temps inclut le temps passé dans les manipulations de la file *Fconnus* par le processus. L'augmentation du nombre de messages a peu d'influence sur ce temps pour des processus *FIFO* parce que la longueur de la file importe peu

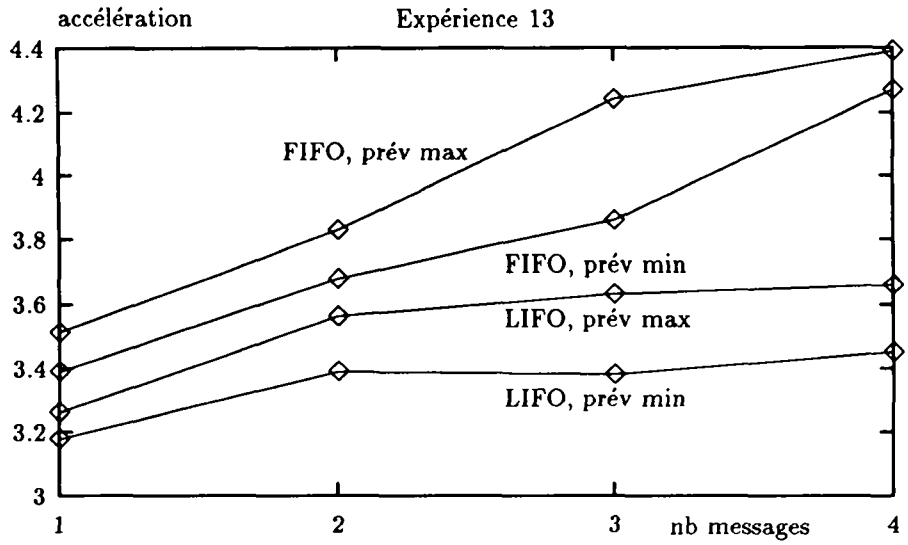

Figure 21 : accélération = $\mathcal{F}$ (nombre de messages) pour le tore

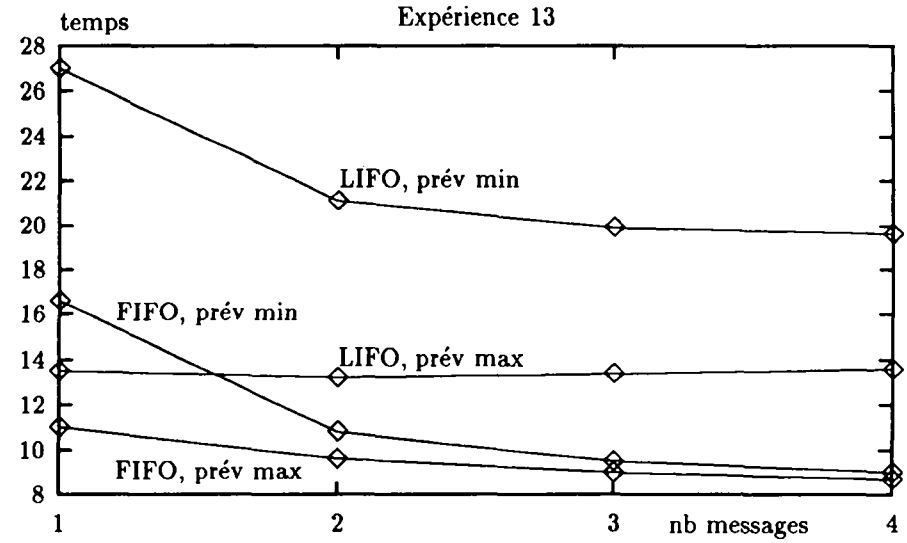

Figure 22 : temps = $\mathcal{F}$ (nombre de messages) pour le tore

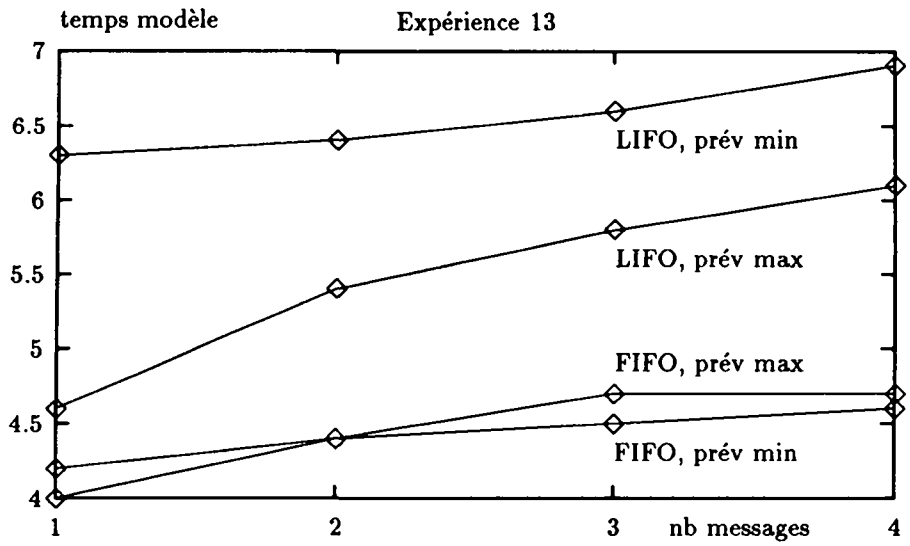

Figure 23 : temps modèle = $\mathcal{F}$ (nombre de messages) pour le tore

dans ce cas, le temps passé dans le modèle augmente de façon importante pour un processus *LIFO* avec une prévision maximale parce qu'il faut parcourir toute la file à chaque fois.

Cet effet ne se fait pas sentir avec des processus *LIFO* et une prévision minimale à cause de la mise en œuvre des blocages / réactivations de processus évoquée en 2.2.3: un processus bloqué en attente de message peut être réactivé inutilement par l'arrivée d'un message *NULL*. La prévision minimum avec un processus *LIFO* implique un grand nombre de messages *NULL*, et donc de nombreux déblocages inutiles. Le temps modèle est alors important avec un seul message, et ensuite l'augmentation du temps de parcours des files dûe à l'augmentation du nombre de messages est compensé par la diminution du nombre de messages *NULL*; c'est ce qui explique la relative stabilité du temps modèle dans ce cas.

Les figures 24 et 25 donnent les temps de simulation sur le réseau à nombre variable d'entrées en fonction du nombre de messages par processus (expériences 14 et 15). On constate également une diminution des temps de simulation quand le nombre de messages augmente, diminution d'autant plus forte que le nombre de canaux d'entrée par processus est grand.

4.2.6 Influence de la topologie

Il est difficile d'évaluer l'impact de la forme du réseau de processus simulés sur les performances du simulateur, car il est difficile de quantifier ce paramètre. Pour ne pas avoir d'interférences avec des problèmes de placement nous n'avons considéré que des réseaux réguliers. Notre étude a porté essentiellement sur le nombre de canaux d'entrée pour chaque processus, car, compte tenu de l'algorithme de contrôle utilisé, cela semble a priori un paramètre important. Un autre facteur, a priori intéressant, le nombre et la taille des circuits dans le graphe, n'a pas été étudié.

La figure 26 donne le temps de simulation pour des réseaux dont le nombre d'entrées varie de 1 à 15, pour des processus *FIFO* et *LIFO* (expérience 16). On constate une augmentation importante du temps de simulation avec le nombre de canaux d'entrée par processus, surtout

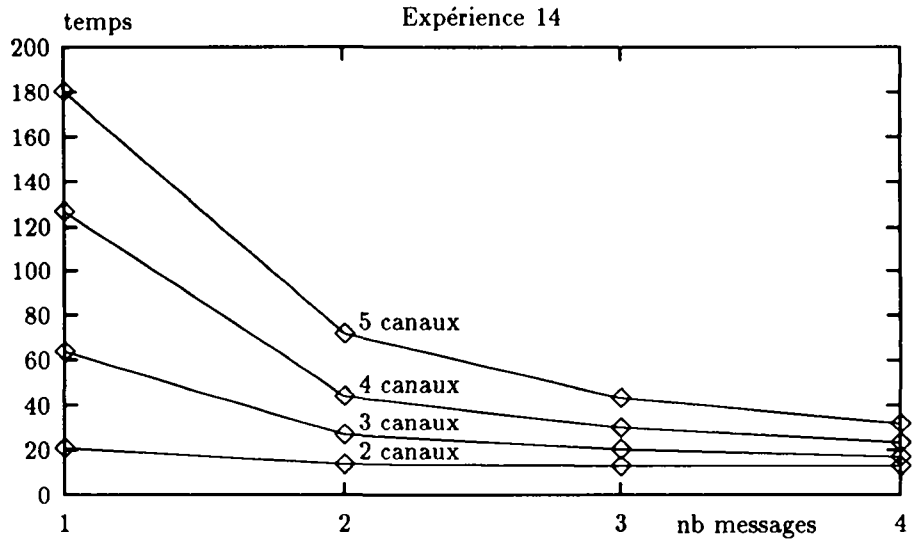

Figure 24 : temps = $\mathcal{F}$ (nombre de messages) pour le *rev*, processus *FIFO*

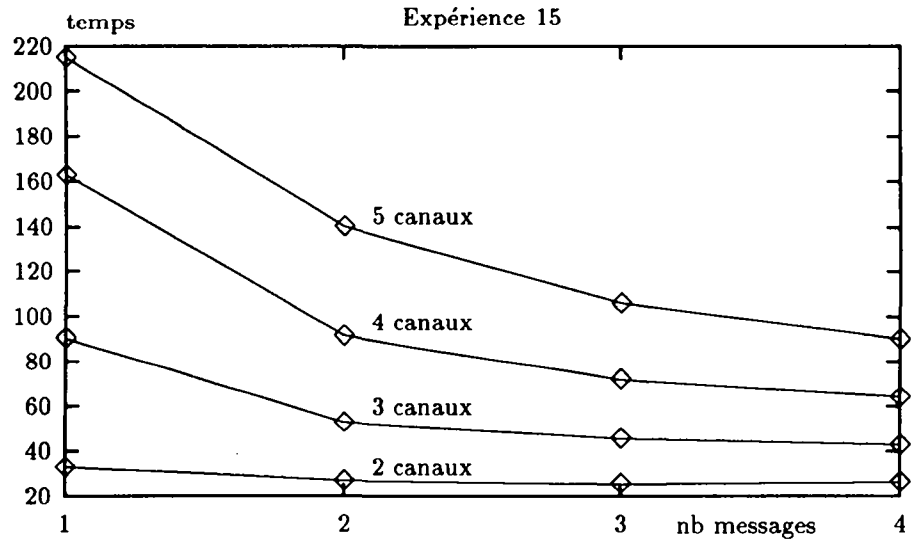

Figure 25 : temps = $\mathcal{F}$ (nombre de messages) pour le *rev*, processus *LIFO*

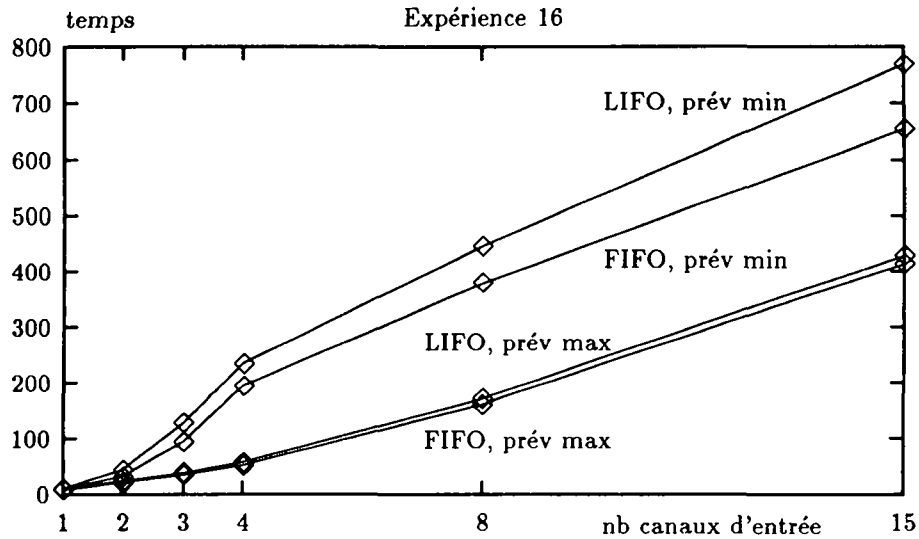

Figure 26 : temps = $\mathcal{F}$ (nombre de canaux d'entrée) pour le *rev*

quand la prévision est mauvaise. Ce phénomène est assez indépendant du comportement des processus (*FIFO* ou *LIFO*). Les figures 27 et 28 donnent respectivement les pourcentages de messages *NULL* et le temps-modèle pour les mêmes expériences. On y constate d'une part une forte augmentation du nombre de messages *NULL*, à peu près indépendante de la prévision, et d'autre part une augmentation du temps-modèle liée fortement à celle-ci. Le nombre élevé de messages *NULL* a deux effets négatifs différents : le noyau passe du temps à les émettre et à les recevoir, mais comme on l'a indiqué précédemment ils peuvent également impliquer dans certains cas des réactivations inutiles des processus. C'est ce phénomène qui explique le temps modèle plus élevé dans le cas de la prévision minimale : dans ce cas un processus doit recevoir plusieurs messages *NULL* avant de pouvoir traiter un message de la simulation.

5 Conclusions

La réalisation du noyau *Floria* nous a permis d'abord de préciser les choix architecturaux qui peuvent se poser lors de la mise en œuvre répartie d'applications liées à un temps virtuel. Un des caractères principaux de l'architecture de *Floria* est la séparation en deux couches, bien délimitées, de la gestion de la communication dans le temps virtuel et du contrôle de l'exécution des processus. Sur ce point elle se distingue nettement de la structure de *PARSEVAL* [14], un simulateur de réseaux de files d'attente développé sur un réseau de transputers. Cette séparation permet de bien isoler le contrôle des contraintes de causalité, qui a fait l'objet des principales études sur la simulation distribuée, et pour lequel des solutions algorithmiques particulières ont été proposées [12]. On peut ainsi facilement essayer telle ou telle variante de ces algorithmes. Cette séparation permet également d'avoir un produit qui n'est pas lié à une classe d'applications particulière (réseau de file d'attente, ...)

L'impossibilité de tirer parti des propriétés du modèle dans l'algorithme de gestion de la

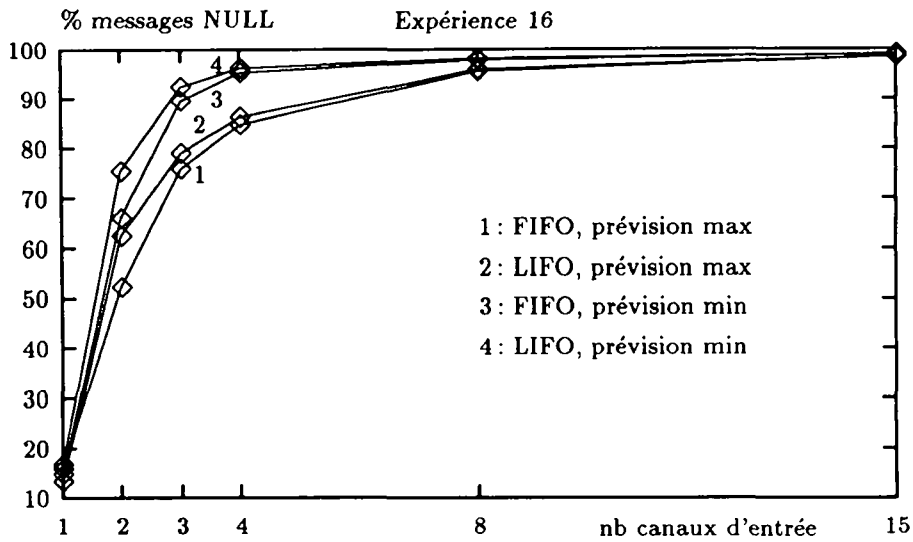

Figure 27 : % *NULL* = $\mathcal{F}$ (nombre de canaux d'entrée) pour le *rev*

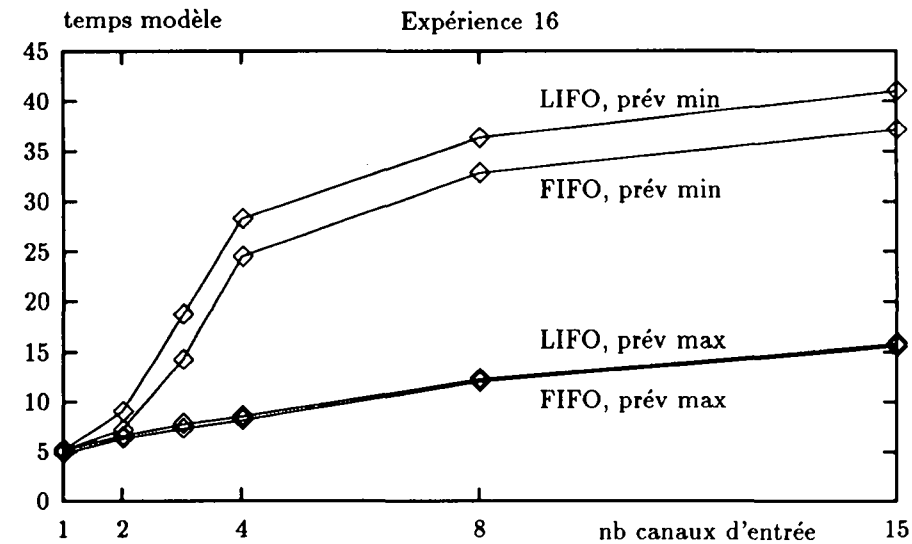

Figure 28 : temps modèle = $\mathcal{F}$ (nombre de canaux d'entrée) pour le *rev*

communication, à cause de cette séparation en deux couches, pouvait faire craindre une dégradation des performances du simulateur. La définition de l'interface externe du noyau *Floria*, et sa mise en œuvre dans la couche de contrôle de l'exécution des processus, permettent cependant d'éviter cet écueil. On a à la fois une interface générale et souple, et une mise en œuvre qui se révèle efficace quand le comportement des processus le permet (processus *FIFO*).

Globalement ces expériences ont montré que l'on pouvait effectivement obtenir une diminution du temps de simulation, en répartissant une simulation avec un algorithme de contrôle de type conservatif. Même si l'efficacité (rapport entre le gain obtenu par la distribution et le nombre de processeurs utilisés) est faible, comme on l'a déjà signalé, l'important est de pouvoir diminuer les temps de simulation. On peut envisager des améliorations à ces algorithmes de contrôle conservatifs : accélérateurs pour éviter les cycles de messages *NULL* répétitifs [11], utilisation de la mémoire partagée entre les processus implantés sur le même site. Ces améliorations n'ont pas encore été réalisées et évaluées, mais la structure modulaire du noyau facilite ce type d'étude.

Par contre, l'augmentation du nombre de canaux d'entrée par processus est un facteur pénalisant pour les simulateurs répartis utilisant les méthodes conservatives. Enfin la qualité de la prévision est d'autant plus importante que les autres caractéristiques du modèle sont défavorables (nombre de canaux d'entrée, charge, nombre de messages, ...).

Les performances de nos simulations réparties sont d'autant meilleures que la charge en calcul est importante et que le nombre de messages est élevé. Or ces deux caractéristiques sont pénalisantes pour les simulations séquentielles. Les simulations qui se prêtent le mieux à une parallélisation sont donc celles qui posent le plus de problèmes en centralisé, c'est à dire justement celles que l'on a envie de distribuer. C'est une indication très encourageante pour ce type de technique.

Nous envisageons d'effectuer une mise en œuvre sur notre noyau d'un sous-ensemble d'*ESTELLE* [8], utilisant un temps virtuel via la clause *DELAY*. Cela nous permettrait d'une part d'utiliser notre noyau dans un cadre plus réaliste, et d'autre part d'étudier les mécanismes de calcul automatique de la prévision sur les canaux d'entrée.

A Annexe : Exemple de programme s'exécutant sur *Floria*

Ce code en C ci-dessous présente la modélisation du tore de serveurs *LIFO* (figure 3) utilisé dans l'expérience 2 du tableau 5. Ce code est divisé en deux parties : la première définit le corps type de chaque processus et le contenu de chaque type de message; la deuxième (définie par la fonction *ModelDescription*) réalise la création et connexion des processus.

```
/*****************************************************/

#include "floria.h"   /* Floria interface          */
#include "random.c"   /* random numbers generators */
#include <stdio.h>

#define      DIMESH    16  /* mesh dimension       */
#define      PROCESSOR  1  /* number of processors */
#define      PHYSLOAD  16  /* physical load        */

/***** message types *******************************/

typedef struct
{
  HeaderType Header; /* First field in all message types */
                     /* This msg type has an empty body  */
} MsgType001, *MsgPtr001;

/***** body of processes ***************************/

void Body000 ()
{
  MsgPtr001 Msg;
  real ServTime;
  int  i,j, MySeed;

  /* random number seed initialization */
  MySeed = 5 * MySelf () + 34;

  /* send a new msg to output 0 */
  Msg = (MsgPtr001) MakeMsg (001);
  SendMsg ((MsgPtr) Msg, 0);

  while (1) {
    /* set lookahead = minimum service time */
    ServTime = 1.0 + (real) EXPN (9.0, &MySeed);
    SetLook (1.0);

    /* take next customer at the end of the input queue */
    if (Empty ())
      (void) WaitMsg (Infinity);
    Msg = (MsgPtr001) Take (Last ());
```

```
    /* virtual workload */
    Hold (ServTime);

    /* physical workload */
    for (i = 1; i <= PHYSLOAD; i++)
      for (j = 1; j <= 50; j++)
        x = 3.1416 + j;

    SendMsg ((MsgPtr) Msg, MsgInput ((MsgPtr) Msg));
  }
}

/***** model description ***************************/

void ModelDescription ()
{
  int i,j,proc;

  /* creation of processes:                    */
  /* Create (GlobalID, NodeNumber, BodyFunc,  */
  /*         StackSize, NbInputs, NbOutputs)  */
  for (i = 1; i <= DIMESH; i++)
    for (j = 1; j <= DIMESH; j++)
      Create (DIMESH * (i-1)+j, ((i-1) * PROCESSOR) / DIMESH,
              Body000, 6000, 2, 2);

  /* connection of channels to build the mesh:        */
  /* Connect (SenderID, OutChan, ReceiverID, InpChan) */
  for (i = 1; i <= DIMESH; i++)
    for (j = 1; j <= DIMESH; j++)
      {
        proc = (i - 1) * DIMESH + j;
        Connect (proc, 0, (i-1) * DIMESH + (j % DIMESH) + 1, 1);
        Connect (proc, 1, (i % DIMESH) * DIMESH + j, 0);
      }

  /* maximum simulation time */
  MaxTime  = 5000.0;
}

/***************************************************/
```

Bibliographie

[1] K. M. Chandy and J Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Comm. of the ACM*, 24(11), Avril 1981.

[2] K.M. Chandy and J. Misra. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Trans. on Soft. Eng., Vol. 5, No 5*, 440–452, September 1979.

[3] S. Cheung. *Distributed computer simulation of a data communication network.* Technical Report CSD-870039, University of California – Los Angeles, July 1987.

[4] R.M. Fujimoto. Lookahead in parallel discrete event simulation. In *Proc. int. conference on parallel processing*, pages 34–41, 1988.

[5] R.M. Fujimoto. *Performance measurement of distributed simulations strategies.* Tech. Report UUCS-87-026a, University of Utah, November 1987.

[6] R.M. Fujimoto. Performance measurements of distributed simulation strategies. In *Proc. SCS conference on Distributed Simulation, San Diego*, pages 14–20, February 1988.

[7] Ph. Ingels, C. Maziero, and M. Raynal. A distributed kernel for virtual time driven applications. In *IEEE International Conference on Computers and Information (ICCI 92), Toronto*, pages 430–433, May 1992. also published as Research Report RR-1605 INRIA-IRISA Rennes, France.

[8] ISO 9074. *Proposed draft addendum to ISO 9074:1989 — Estelle tutorial.* ISO 9074:1989 ISO/IECJTC1/SC21/F60.

[9] D. Jefferson. Virtual time. *ACM Toplas, Vol. 7, No 3*, 404–425, July 1985.

[10] D. Jefferson et al. Distributed simulation and the time-warp operating system. In 11^{th} *ACM Symposium on Operating Systems Principles, Austin - Texas*, pages 77–93, Operating System Review, ACM Press, November 1987.

[11] Muhlhauser M. Using distributed simulation for distributed application development. In *The 21st Annual Simulation Symposium, Tampa - Florida*, pages 189–206, March 1988.

[12] J. Misra. Distributed discrete-event simulation. *Computing Surveys, Vol. 18, No 1*, 39–65, March 1986.

[13] D. Potier and M. Veran. Qnap2: a portable environment for queueing systems modelling. In *Int. Conf. on Modelling Technics and Tools for Performance Evaluation*, 1984.

[14] H. Rakotoarisoa and Ph. Mussi. *PARSEVAL: parallelisation sur reseaux de transputers de simulations pour l'evaluation de performances.* Research report RT-131, INRIA, September 1991.

LISTE DES DERNIERES PUBLICATIONS INTERNES PARUES A L'IRISA

PI 668 TWO COMPLEMENTARY NOTES ON SKEWED-ASSOCIATIVE CACHES
André SEZNEC
Juillet 1992, 10 pages.

PI 669 PARALLELISATION D'UN ALGORITHME DE DETECTION DE MOUVEMENT SUR UNE ARCHITECTURE MIMD
Fabrice HEITZ, Sergui JUFRESA, Etienne MEMIN, Thierry PRIOL
Juillet 1992, 34 pages.

PI 670 UN RESEAU SYSTOLIQUE INTEGRE POUR LA CORRECTION DE FAUTES DE FRAPPE
Dominique LAVENIER
Juillet 1992, 120 pages.

PI 671 EARLY WARNING OF SLIGHT CHANGES IN SYSTEMS AND PLANTS WITH APPLICATION TO CONDITION BASED MAINTENANCE
Qinghua ZHANG, Michèle BASSEVILLE, Albert BENVENISTE
Juillet 1992, 32 pages.

PI 672 ORDRES REPRESENTABLES PAR DES TRANSLATIONS DE SEGMENTS DANS LE PLAN
Vincent BOUCHITTE, Roland JEGOU, JeanXavier RAMPON
Juillet 1992, 8 pages.

PI 673 AN EXCEPTION HANDLING MECHANISM FOR PARALLEL OBJECT-ORIENTED PROGRAMMING
Valérie ISSARNY
Août 1992, 36 pages.

PI 674 A CALCULUS OF GAMMA PROGRAMS
Chris HANKIN, Daniel LE METAYER, David SANDS
Juillet 1992, 32 pages.

PI 675 EVALUATION DES PERFORMANCES D'UN NOYAU DE SIMULATION REPARTIE
Philippe INGELS, Carlos MAZIERO
Septembre 1992, 36 pages.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249 - 6399